SUBSTITUTE SPECIFICATION

CLEAN COPY

Attorney Docket No. SUN1P814/P5417

PATENT APPLICATION

IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS

Inventors: 1. Stepan SOKOLOV
34832 Dorado Common
Fremont, CA 94555
Citizen of Ukraine

2. David WALLMAN
777 S. Mathilda Ave. #266
Sunnyvale, CA 94087
Citizen of Israel

Assignee: Sun Microsystems, Inc.
901 San Antonio Road
M/S PAL01-521
Palo Alto, CA 94303

BEYER WEAVER & THOMAS, LLP
P.O. Box 778
Berkeley, CA 94704-0778
Telephone (650) 961-8300

IMPROVED FRAMEWORKS FOR LOADING AND EXECUTION OF OBJECT-BASED PROGRAMS

CROSS-REFERENCE TO RELATED APPLICATIONS

This application is related to U.S. Patent Application No. 09/703,361, filed October 31, 2000 (Atty. Dkt. No. SUN1P809/P5500), entitled "IMPROVED FRAMEWORKS FOR INVOKING METHODS IN VIRTUAL MACHINES", filed concurrently herewith, and hereby incorporated herein by reference.

This application is related to U.S. Patent Application No. 09/703,356, filed October 31, 2000 (Atty. Dkt. No. SUN1P810/P5510), entitled "IMPROVED METHODS AND APPARATUS FOR NUMERIC CONSTANT VALUE INLINING IN VIRTUAL MACHINES", filed concurrently herewith, and hereby incorporated herein by reference.

BACKGROUND OF THE INVENTION

The present invention relates generally to object-based high level programming environments, and more particularly, to frameworks for loading and execution of portable, platform independent programming instructions within a virtual machine.

Recently, the Java™ programming environment has become quite popular. The Java™ programming language is an language that is designed to be portable enough to be executed on a wide range of computers ranging from small devices (e.g., pagers, cell phones and smart cards) up to supercomputers. Computer programs written in the Java™ programming language (and other languages) may be compiled into Java™ virtual machine instructions (typically referred to as Java™ bytecodes) that are suitable for execution by a Java™ virtual machine implementation.

The Java™ virtual machine is commonly implemented in software by means of an interpreter for the Java™ virtual machine instruction set but, in general, may be software, hardware, or both. A particular Java™ virtual

machine implementation and corresponding support libraries, together constitute a Java™ runtime environment.

Computer programs in the Java™ programming language are arranged in one or more classes or interfaces (referred to herein jointly as classes or class files). Such programs are generally platform, i.e., hardware and operating system, independent. As such, these computer programs may be executed, unmodified, on any computer that is able to run an implementation of the Java™ runtime environment. A class written in the Java™ programming language is compiled to a particular binary format called the "class file format" that includes Java™ virtual machine instructions for the methods of a single class. In addition to the Java™ virtual machine instructions for the methods of a class, the class file format includes a significant amount of ancillary information that is associated with the class. The class file format (as well as the general operation of the Java™ virtual machine) is described in some detail in The Java™ Virtual Machine Specification by Tim Lindholm and Frank Yellin (ISBN 0-201-31006-6), which is hereby incorporated herein by reference.

Conventionally, when a class file is loaded into the virtual machine, the virtual machine essentially makes a copy of the class file for its internal use. The virtual machine's internal copy is sometimes referred to as an "internal class representation." In conventional virtual machines, the internal class representation is typically almost an exact copy of the class file and it replicates the entire Constant Pool. This is true regardless of whether multiple classes loaded into the virtual machine reference the same method and thus replicate some (or much) of the same Constant Pool information. Such replication may, of course, result in an inefficient use of memory resources. In some circumstances (particularly in embedded systems which have limited memory resources) this inefficient use of memory resources can be a significant disadvantage.

As described in The Java™ Virtual Machine Specification, one of the structures of a standard class file is known as the "Constant Pool." The Constant Pool is a data structure that has several uses. One of the uses of the Constant Pool that is relevant to the present invention is that the Constant

Pool contains the information that is needed to resolve various Java™ Instructions, for example, a method invocation instruction that can be invoked by any of the methods within the class. Fig. 1A (which may be familiar to those skilled in the art) is a representation of a Constant Pool section of a class file that contains the information necessary to uniquely identify and locate a particular invoked method.

Additionally, conventional virtual machine interpreters decode and execute the virtual machine instructions (Java™ bytecodes) one instruction at a time during execution, e.g., "at runtime." To execute a Java™ instruction, typically, several operations have to been performed to obtain the information that is necessary to execute the Java™ instruction. For example, to invoke a method referenced by a Java™ bytecode, the virtual machine must make several operations to access the Constant Pool simply to identify the information necessary to locate and access the invoked method.

To illustrate, Fig. 1B shows an exemplary conventional Java™ bytecode stream 150 including virtual machine instructions (typically referred to as Java™ bytecodes) that are suitable for execution by a conventional Java™ virtual machine. The first bytecode, Java™ bytecode 152 represents an arithmetic Java™ "Add" command with no associated parameters. The second bytecode, Java™ bytecode 154 represents a Java™ "iconst" instruction (load an integer constant on the stack). The Java™ bytecodes 156 and 157, CP-IndexA and CP-IndexB, respectively represent the first and second bytes of an index to the constant pool for the integer constant. It should be noted that Java™ bytecodes 152, 154 and 156 collectively represent a Java™ "iconst" instruction. In order to execute the Java™ Instruction iconst, at run time, an index to the Constant Pool is constructed from the CP-IndexA and CP-IndexA. Once an index to the Constant Pool has been determined, the appropriate structures in the Constant Pool have to be accessed so that the appropriate constant value can be determined. Accordingly, the Java™ instruction "iconstant" is executed but only after performing several operations at run time. As can be appreciated from the example above, the execution of a relatively simple instruction such as loading a constant value can take a significant amount of run time.

Furthermore, execution of relatively more complex Java™ instructions requires even more processing than noted above. For example, in order to conventionally execute an invoke method command 158, the constant pool has to be accessed several times at run time just to obtain the symbolic representation of information associated with the method. The CP_IndexC and CP_IndexD are indexes to a constant pool where information relating to the method including its parameter values needed for execution can be found. Accordingly, a CP_IndexD associated with the method instruction 158 is typically an index to an entry into the Constant Pool wherein that entry itself provides another pair of indexes to other entries in the Constant Pool, and so forth. Thus, execution of an invoke method command would require a significant amount of processing at run time (e.g., accessing the Constant Pool several times to obtain symbolic representation relating to the method). As will be appreciated, this conventional technique is an inefficient approach that may result in significantly longer execution times. In addition, once the symbolic information relating to the method has been obtained, there may be a need to convert the symbolic information into an internal representation at run time. Again, this is an inefficient approach.

Accordingly, there is a need for improved frameworks for loading and execution of portable, platform independent programming instructions within a virtual machine.

SUMMARY OF THE INVENTION

To achieve the foregoing and other objects of the invention, improved frameworks for loading and execution of portable, platform independent programming instructions within a virtual machine will be described. One aspect of the present invention seeks to provide a mechanism that will generally improve the runtime performance of virtual machines by eliminating the need to always traverse a constant pool at runtime to execute a Java™ instruction. In effect, the described system contemplates doing some extra work during the loading of a class into a virtual machine by obtaining the information from the constant pool during loading and representing that information in a form that can be used more efficiently at runtime.

In other aspects of the invention, specific data structures that are suitable for use within a virtual machine and methods for creating such data structures are described. In one embodiment, an enhanced Java™ bytecode representation having a pair of Java™ bytecode streams is disclosed. The enhanced Java™ bytecode has a Java™ code stream suitable for storing various Java™ commands as bytecodes within a code stream. A Java™ data stream is used to store the data parameters associated with the Java™ commands in the code stream. As will be appreciated, the invention allows for representation of actual parameter values, or references to actual parameter values, in the code stream. Accordingly, data parameters can be provided for efficient execution of Java™ instructions without requiring further processing of Constant Pools at run time. As a result, the performance of Java™ complaint virtual machine can be enhanced.

The invention can be implemented in numerous ways, including a system, an apparatus, a method or a computer readable medium. Several embodiments of the invention are discussed below.

As a method of creating data structures suitable for use by a virtual machine to execute computer instructions, one embodiment of the invention includes converting a stream, having commands and data associated with the commands, into a pair of streams for use in the virtual machine. The pair of streams includes a code stream and a data stream for data associated with the commands in the code stream.

As a data structure for containing computer executable commands and data associated with the computer executable commands, the data structure suitable for use by a virtual machine in an object oriented programming environment, one embodiment of the invention includes a code portion having one or more computer executable commands, and a data stream having data corresponding to the one or more computer executable commands.

As a method of executing computer instructions on a virtual machine, one embodiment of the invention includes the acts of: fetching a command associated with a computer instruction from a code stream; determining whether the command has an associated parameter; fetching from a data stream the associated parameter of the command when that command has

an associated parameter; and executing the command with the associated parameters after the associated parameter of the commands have been fetched.

As a method of creating data structures suitable for use by a virtual machine to execute Java™ instructions, another embodiment of the invention includes converting a Java™ bytecode into a pair of Java™ bytecode streams suitable for use by the virtual machine. The pair of Java™ bytecode streams being a Java™ code stream that includes Java™ commands and a Java™ data stream that includes the data associated with the commands in the Java™ code stream.

Other aspects and advantages of the invention will become apparent from the following detailed description, taken in conjunction with the accompanying drawings, illustrating by way of example the principles of the invention.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will be readily understood by the following detailed description in conjunction with the accompanying drawings, wherein like reference numerals designate like structural elements, and in which:

Fig. 1A is a representation of a Constant Pool section of a Java™ class file.

Fig. 1B shows an exemplary conventional Java™ bytecode stream including virtual machine instructions.

Fig. 2 is a block diagram representation of Java™ bytecode streams in accordance with one embodiment of the invention.

Fig. 3 illustrates an execution method for executing Java™ bytecode instructions in accordance with one embodiment of the invention.

Fig. 4 illustrates an exemplary bytecode stream generation method in accordance with one embodiment of the invention.

Fig. 5 illustrates an exemplary loading method for loading bytecodes of a class file in accordance with one embodiment of the invention.

Fig. 6 illustrates a processing method for processing a Java™ Load constant command in accordance with one embodiment of the invention.

Fig. 7 illustrates an execution method for executing a Java™ Load Constant command in accordance with one embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

As described in the background section, the Java™ programming environment has enjoyed widespread success. Therefore, there are continuing efforts to extend the breadth of Java™ compatible devices and to improve the performance of such devices. One of the most significant factors influencing the performance of Java™ based programs on a particular platform is the performance of the underlying virtual machine. Accordingly, there have been extensive efforts by a number of entities to provide improved performance Java™ compliant virtual machines. In order to be Java™ compliant, a virtual machine must be capable of working with Java™ classes, which have a defined class file format. Although it is important that any Java™ virtual machine be capable of handling Java™ classes, the Java™ virtual machine specification does not dictate how such classes are represented internally within a particular Java™ virtual machine implementation.

The Java™ class file format utilizes the constant pool construct to store the information necessary to execute a Java™ instruction at run time. However, as suggested above, traversing the constant pool at runtime to obtain the information necessary to execute Java™ instructions is not particularly efficient. One aspect of the present invention seeks to provide a mechanism that will generally improve the runtime performance of virtual machines by eliminating the need to always traverse a constant pool at runtime to execute a Java™ instruction. Accordingly, a significant amount of work is conventionally performed at runtime. In effect, the described system

contemplates doing some extra work during the loading of a class into a virtual machine by obtaining the information from the constant pool during loading and representing that information in a form that can be used more efficiently at runtime

In other aspects of the invention, specific data structures that are suitable for use within a virtual machine and methods for creating such data structures are described. In one embodiment, an enhanced Java™ bytecode representation having a pair of Java™ bytecode streams is disclosed. The enhanced Java™ bytecode has a Java™ code stream suitable for storing various Java™ commands as bytecodes within a code stream. A Java™ data stream of the enhanced Java™ bytecode representation is used to store the data parameters associated with the Java™ commands in the code stream. As will be appreciated, the invention allows for representation in the code stream of actual parameter values, or references to actual parameter values. Accordingly, data parameters can be provided for efficient execution of Java™ instructions without requiring further processing of Constant Pools at run time. As a result, the performance of Java™ complaint virtual machine can be enhanced.

Furthermore, if the invention is implemented together with other improvements as described in concurrently filed, co-pending Application No. U.S. Patent Application No. 09/703,361, filed October 31, 2000 (Atty. Dkt. No. SUN1P809/P5500), entitled "IMPROVED FRAMEWORKS FOR INVOKING METHODS IN VIRTUAL MACHINES", the constant pool may not even need to be copied into the virtual machine's in order to invoke methods. Thus, the use of the invention in conjunction with the improved techniques provided as described in U.S. Patent Application No. 09/703,361, filed October 31, 2000 (Atty. Dkt. No. SUN1P809/P5500), entitled "IMPROVED FRAMEWORKS FOR INVOKING METHODS IN VIRTUAL MACHINES" has the potential in many circumstances to even further improve the performance of the virtual machines.

Embodiments of the invention are discussed below with reference to Figs. 2 - 7. However, those skilled in the art will readily appreciate that the

detailed description given herein with respect to these figures is for explanatory purposes as the invention extends beyond these limited embodiments.

Fig. 2 is a block diagram representation of Java™ bytecode streams 200 in accordance with one embodiment of the invention. The Java™ bytecode streams 200 can be, for example, implemented as a data structure embodied in a computer readable medium that is suitable for use by a virtual machine. As shown in Fig. 2, the Java™ bytecode streams 200 includes a pair of Java™ bytecode streams, namely, a Java™ code stream 202 and a Java™ data stream 204. The code stream 202 includes various Java™ commands 206, 208, and 210. The code stream 206 represents a Java™ command A which does not have any parameters associated with it. On the other hand, code streams 208 and 210 represent respectively Java™ commands B and C with associated data parameters which are represented in the Java™ data stream 204. The Java™ bytecode 212 represents data B which is the data parameters corresponding to the Java™ command B of Java™ code stream 202. It should be noted that the Java™ command B and data B together represent a Java™ instruction corresponding to a Java™ command B with its associated data. Similarly, the Java™ bytecodes 214 and 216 represent respectively data C1 and C2 which collectively represent the data parameters corresponding to the Java™ command C of Java™ code stream 202. Accordingly, the Java™ command C, data C1 and data C2 together represent a Java™ instruction corresponding to a Java™ command C with its associate data represented in bytecodes 214 and 216.

As noted above, the Java™ bytecode 200 streams are suitable for use by a virtual machine. Referring now to Fig. 3, an exemplary execution method 300 for executing Java™ bytecode instructions is disclosed. The execution method 300 can be used, for example, in conjunction with a pair of Java™ bytecode streams, for example, the Java™ code stream 202 and Java™ data stream 204. Initially, at operation 302, the next command in the code stream is fetched. For example, referring back to Fig. 2, the next command fetched can be the bytecode 206 of the Java™ code stream 202. Next, at operation

304, a pointer to the Java™ code stream is incremented to point to the next command in the Java™ code stream. Referring again to Fig. 2, a pointer to the Java™ code stream 206 is incremented to point to the next command in the Java™ code stream, foe example, the bytecode 206 (Java™ command B). Following incrementing the pointer at operation 304, the execution method 300 proceeds to operation 306 where a determination is made as to whether the fetched command has an associated parameter. If it is determined at operation 306 that the fetched command does not have an associated parameter, for example, in the case of Java™ command A of Fig. 2, the execution method 300 proceeds directly to operation 312 where the fetched command (with no associated parameter) is executed. Thereafter, at operation 314 a determination is made as to whether there are more commands in the Java™ code stream to execute. If it is determined at operation 314 that there are no more commands to execute, the execution method 300 ends. However, if it is determined at operation 314 that there is at least one more command to execute, that execution method 300 proceeds back to operation 302 where the next command in the code stream is fetched.

On the other hand, if it is determined at operation 306 that the fetched command has an associated parameter, the execution method 300 proceeds to operation 308 where the parameter values associated with the fetched command are fetched from the data stream. Referring back to Fig 2., for example, in the case when the fetched command is the Java™ command B of bytecode 208, the parameter values associated with Java™ command B, namely, data B of the bytecode 212 is fetched. Similarly, in the case of the Java™ command C (bytecode 210), data C1 and data C2 (bytecodes 214 and 216) would be fetched at operation 308. As will be appreciated, the invention allows for parameter values in the data stream which can represent actual parameter values (or references to actual parameter values). Accordingly, the data parameters in the data code can be used to execute instructions without requiring further processing of Constant Pool.

After the appropriate data is fetched, at operation 310, the pointer to the data command stream is incremented to point to the appropriate position

in the data stream. As will be appreciated, the pointer to the data stream can be updated based on the command type (i.e., based on the number of bytecodes appropriated for the data associated with the command). When the command and the appropriate parameter values have been fetched, the execution method 300 proceeds to operation 314 where the fetched command is executed. Finally, at operation 314 a determination is made as to whether there are more commands in the Java™ code stream to execute. If it is determined at operation 314 that there are no more commands to execute, the execution method 300 ends. However, if it is determined at operation 314 that there is at least one more command to execute, that execution method 300 proceeds back to operation 302 where the next command in the code stream is fetched.

As noted above, the invention, among other things, provides for a pair of Java™ bytecode streams suitable for use by a virtual machine, for example, the code stream 202 and a Java™ data stream 204. Referring now to Fig. 4, an exemplary bytecode stream generation method 400 for generation of an enhanced bytecode representation is disclosed. The bytecode stream generation can be used, for example, to generate a pair of bytecode streams from a conventional Java™ bytecode stream 150 of Fig. 1. Initially, at operation 402, the next bytecode representing a command is read. Next, at operation 404, a bytecode representing the command is written into an entry of a code stream, for example, the Java™ code stream 202 of Fig. 2. Following operation 404, a determination is made at operation 406 as to whether the command has an associated parameter. If it is determined at operation 406 that the command does not have any associated parameters, the bytecode stream generation method 400 proceeds directly to operation 410 where a determination is made as to whether there are more bytecodes to read. If there are no more bytecodes to process, the bytecode stream generation method 400 ends. However, if there is at least one more bytecodes to read, the bytecode stream generation method 400 proceeds to operation 402 where the next bytecode representing a command is read.

On the other hand, when it is determined at operation 406 that the command has associated parameters, the bytecode stream generation

method 400 proceeds to operation 408 where the associated parameters are read and processed. As will be appreciated, the processing of the associated parameters can be performed based on the command type in accordance with one aspect of the invention. This processing will be further illustrated in Fig. 5 below. After appropriate processing of associated parameters, at operation 409, an appropriate value is written to a data stream, for example, the Java™ data stream of Fig. 2. Next, the bytecode stream generation method 400 proceeds to operation 410 where a determination is made as to whether there are more bytecodes to read. if there is at least one more bytecodes to read, the bytecode stream generation method 400 proceeds to operation 402 where the next bytecode representing a command is read. However, if there are no more bytecodes to process, the bytecode stream generation method 400 ends.

Fig. 5 illustrates an exemplary loading method 500 for loading bytecodes of a class file in accordance with one embodiment of the invention. For the sake of illustration, in the described embodiment, the loading is used in a Java™ runtime environment to load a Java™ class file. The class file can include various Java™ instructions (Java™ commands and associated data parameters) . Typically, this would be accomplished by a Java™ class loader. It should be noted that in the described embodiment the Java™ commands are written into a Java™ code stream. Accordingly, the loading method 500 illustrates loading operations that can be performed to provide the corresponding Java™ code stream for the Java™ code stream.

Initially, at operation 502, a determination is made as to whether the Java™ command is a load constant command. If it is determined at operation 502 that the Java™ command is a load constant command, the loading method 500 proceeds to operation 504 where the load constant command is processed in accordance with one aspect of the invention. For example, such processing can be performed as illustrated below in Fig. 6.

On the other hand, if it is determined at operation 502 that the Java™ command is not a load constant command, the loading method 500 proceeds to operation 506 where a determination is made as to whether the Java™ command is an invoke method command. If it is determined at operation 506

that the Java™ command is an invoke method command, the loading method 500 proceeds to operation 508 where the invoke method command is processed in accordance with another aspect of the invention. In a preferred embodiment, the invoke method command is processed to create a reference cell which provides the information necessary to invoke the method at run time. For example, such processing can be performed as illustrated in co-pending U.S. Patent Application No. 09/703,361, filed October 31, 2000 (Atty. Dkt. No. SUN1P809/P5500), entitled "IMPROVED FRAMEWORKS FOR INVOKING METHODS IN VIRTUAL MACHINES". Accordingly, in the described embodiment, the address of the reference cell corresponding to the invoke method command is written into the data stream.

However, If it is determined at operation 506 that the Java™ command is not an Invoke method command, the loading method 500 proceeds to operation 512 where a determination is made as to whether the Java™ command is a Jump command. If it is determined at operation 512 that Java™ command is a Jump command, the appropriate code and data stream offsets of the Jump command are written into the data stream entry corresponding to the jump.

On the other hand, if it is determined at operation 512 that the Java™ command is not a jump command, the loading method 500 proceeds to operation 516 where a determination is made as to whether the Java™ command is an instantiation command. As will be appreciated, if it is determined at operation 516 that the Java™ command is an instantiation command, the Constant pool information for the instantiation command can be processed at operation 518. Following operation 518, in the described embodiment, the appropriate type-descriptor is written into the data stream at operation 520. However, if it is determined at operation 516 that the Java™ command is not an instantiation command, the loading method 500 proceeds to operation 522 where it is determined whether the Java™ command is a Get/Put field command.

As will be appreciated, if it is determined at operation 522 that the Java™ command is a Get/Put field command, the Constant pool information for the instantiation command can be processed at operation 518. For

example, a reference cell which provides the information necessary to execute the Get/Put field command at run time can be generated. Accordingly, following operation 524, the address of a reference cell corresponding to the Get/Put field command can be written into the data stream at operation 526. The loading method 500 ends if it is determined at operation 522 that the Java™ command is not a Get/Put field command.

Fig. 6 illustrates a processing method 600 for processing a Java™ Load constant command in accordance with one embodiment of the invention. The processing method 600 represents operations that can be performed, for example, at operation 508 of Fig. 5. It should be noted that the processing method 600 can be used to convert a conventional representation of a Load constant command with its associated parameter (e.g., bytecodes 154, 156, and 157 of Fig 1). Accordingly, the processing method 600 can be used to represent the Load constant command and its associated parameter respectively into a Java™ code stream and a Java™ data stream, for example, the Java™ code stream 202 and Java™ data stream 204 of Fig. 2.

Initially, at operation 602, the first bytecode of the Constant Pool index (e.g., bytecode 156 of Fig. 1) is fetched. Next, at operation 604, the second bytecode of the Constant Pool index (e.g., byte code 157 of Fig. 1) is fetched. As will be appreciated, at operation 606, the Constant Pool index is constructed from the first and second fetched bytes. After construction of the Constant Pool index, appropriate data structures of the Constant Pool are read at operation 608. Next, at operation 610, the corresponding constant value is determined based on the constant information read at operation 608. As will appreciated, other operations can be performed, for example, if necessary, byte alignment can be performed at operation 610. Finally, at operation 612 the constant value determined at operation 610 is written into a Java™ data stream, for example, the Java™ data stream 204 of Fig. 2. It should be noted that corresponding Java™ Load constant command is written into an appropriate entry of a Java™ code stream, for example, the Java™ code stream 202 of Fig. 2.

Fig. 7 illustrates an execution method 700 for executing a Java™ Load Constant command in accordance with one embodiment of the invention. It

should be noted that the execution method 700 can be implemented in a virtual machine in conjunction with an enhanced bytecode representation to allow more efficient run time execution of a Java™ Load Constant command. For example, the processing method 600 can be used to generate the bytecode streams 200 having the Java™ code stream 202 and Java™ data stream 204. Accordingly, The Java™ Load Constant command can be stored in the code stream while the corresponding Constant parameters value are stored in the data stream.

Initially, at operation 702, the Java™ Load Constant command is fetched from the code stream. For example, referring back to Fig. 2, the bytecode 206 of the Java™ code stream 206 (Java™ command A), representing a Java™ Load Constant command is fetched from the Java™ code stream 202. Next, at operation 704, a pointer to the Java™ code stream is incremented to point to the next command in the Java™ code stream. Referring again to Fig. 2, a pointer to the Java™ code stream 206 is incremented to point to the next command in the Java™ code stream, namely, code stream 206 (Java™ command B). Following incrementing the pointer at operation 704, the execution method 700 proceeds to operation 706 where it is determined how many bytecodes corresponding to the Constant parameter value needs to be fetched from the Java™ data stream. As will appreciated this determination can be made based on the Constant's type (e.g., double, integer, float, etc.). Accordingly, at operation 708, the appropriate number of bytes which correspond to the Constant parameter value are fetched.

After the appropriate constant value is fetched, at operation 710, the pointer to the data command stream is incremented to point to the appropriate position in the data stream. As will be appreciated, the pointer to the data stream can be updated based on the type of the Load constant command (i.e., based on the number of bytecodes appropriated for the data associated with the command). Finally, at operation 712, the Constant Load command is executed, for example, the appropriate Constant value command in pushed on the stack.

The many features and advantages of the present invention are apparent from the written description, and thus, it is intended by the appended

claims to cover all such features and advantages of the invention. Further, since numerous modifications and changes will readily occur to those skilled in the art, it is not desired to limit the invention to the exact construction and operation as illustrated and described. Hence, all suitable modifications and equivalents may be resorted to as falling within the scope of the invention.

What is claimed is: